A Tutorial on Differential Inverse Kinematics

Sam Williams

***Abstract*—Setting the position and orientation of a robotic manipulator's end effector requires solving for the joint angles which achieve that pose. This problem, inverse kinematics, has many existing solutions. Differential inverse kinematics is one such method which solves inverse kinematics by computing the Jacobian of the system which directly relates joint velocities to end effector velocities. Differential inverse kinematics enables path following to a desired end pose and provides insight into the end effector's ease of motion. This tutorial aims to clearly explain the derivation and implementation of three different methods for differential inverse kinematics, provide clear distinction for the benefits and drawbacks of each method, and provide practical advice for creating a controller using any of these methods.**

I. Introduction

Any application that needs to set the position and orientation of a robotic manipulator's end effector necessarily has to solve inverse kinematics (IK) to get the joint angles which achieve that pose. Additionally, the path the end effector takes to that location is often important. Paths which needlessly waste time or collide with obstacles should be avoided. Differential inverse kinematics solves this exact problem, enabling IK solutions that follow paths. Differential inverse kinematics works by computing the Jacobian of the end effector which relates joint velocities to end effector velocities. Differential inverse kinematics allows directly setting approximate end effector velocities to achieve a desired path. If we want to find the shortest Cartesian path to a desired pose, we can set the velocity as that vector. If we want to achieve a desired pose while avoiding obstacles, we can set the velocity to go around them. Differential inverse kinematics is ideal for any problem which needs to set the pose of an end effector while maintaining a sensible path to get to that pose. Differential inverse kinematics also provides tools to understand when the manipulator is unable to move in certain directions, analyze our expected error, and get a sense of how much freedom of motion there is in any given pose.

Other documents which explain differential inverse kinematics exist, [1][2][3] are just a few. Differential inverse kinematics is a theoretically motivated method which can appear opaque at first glance. This document attempts to assume less prior knowledge than other references and provide a more complete view of the method. Additionally, many important details for implementing one of these methods are often swept under the rug. Computing the Jacobian for a robotic manipulator is simple given the homogeneous transforms of the system but requires explanation. The question of how to set a desired velocity, especially angular velocity, to follow a path is surprisingly difficult. We must remember the Jacobian is a linear approximation of the forward kinematics at the current configuration, the solved joint angles need to be scaled down to an appropriate step size. Finally, we must consider different possible convergence checks. This tutorial aims to iron out all the details in implementing differential inverse kinematics. Once we've worked through these issues, we will have a method for inverse kinematics which allows us to follow desired paths and provides great insight into our system.

II. Preliminaries

In order to present the methods to solve differential inverse kinematics we must cover some preliminaries. First, we clearly define forward and inverse kinematics. We then solve the linear and angular velocity of the end effector due to the change in angle from each joint. Then we use those velocities to define and solve for the Jacobian matrix of the system. We take a brief aside for the Singular Value Decomposition to explain its properties when applied to the Jacobian. Finally, we solve for the linear and angular velocities necessary to achieve a desired pose.

A. Forward and Inverse Kinematics

The configuration of a robotic manipulator is specified by joint values. We will assume all joints are revolute, making the joint values the column vector $\theta = [\theta_0, ..., \theta_{n-1}]^T$. Forward kinematics takes these joint values and outputs the homogeneous transform of the end effector.

$$T_n^0 = f(\theta) \tag{1}$$

This homogeneous transform expresses the position and orientation of the end effector in the base frame and is composed of the orthonormal rotation matrix R_n^0 and the translation u.

$$T_n^0 = \begin{bmatrix} R_n^0 & u \\ 0 & 1 \end{bmatrix}$$

The forward kinematics function f can be computed through multiple methods. We will assume it is computed following the Denavit-Hartenberg convention for simplicity. Inverse kinematics takes the inverse of f to compute the joint angles required to reach a desired transform.

$$f^{-1}(T_n^0) = \theta \tag{2}$$

As f is non-linear, subsequent joints depend on the previous joint positions, taking this inverse is difficult. Analytical solutions exist for manipulators up to 6 degrees of freedom. The inverse kinematics function for manipulators with more degrees of freedom leads to an under-determined system of equations with either no or infinite solutions. Again for simplicity, we will consider manipulators with 6 degrees of freedom to have a fully-determined system, but the methods presented in this tutorial will apply to manipulators with arbitrary degrees of freedom.

B. Linear and Angular Velocities of a End Effector

The linear velocity of an end effector due to each joint is simply the joint angle derivative of the position.

$$v = \frac{\partial}{\partial \theta} u$$

The contribution to the linear velocity from each joint is linear, i.e. we can solve for the contribution from each joint and take the entire sum as the linear velocity of the end effector. The contribution from joint i depends on the axis of rotation z_i, the end effector position u, the joint position p_i, and the rate of change in angle θ'_i. Note the axis of rotation is the z-axis since we are following the Denavit-Hartenberg convention.

$$v = \Sigma_{i=0}^{n-1} [z_i \times (u - p_i)] \theta'_i \tag{3}$$

Angular velocity is a little bit more tricky. Angular velocity describes some speed of rotation with respect to an axis, yet our rotation axis will change as we move and rotate the end effector. Additionally, the orientation of the end effector is provided as a rotation matrix. We cannot interpolate rotation matrices to apply some partial rotation between two frames. We can however use the definition of angular velocity to get the contribution from each joint. The contribution from each joint to the angular velocity is linear.

$$\omega = \Sigma_{i=0}^{n-1} z_i \theta'_i \tag{4}$$

C. The Jacobian Matrix

The Jacobian matrix is a linear approximation of the forward kinematics of our robotic manipulator at the current configuration. It defines a linear surface that gives an approximate change in end effector pose for a change in joint angle configuration. Formally, the Jacobian matrix relates the joint velocities θ' to the linear and angular velocities v, ω.

$$s = \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} J_v \\ J_\omega \end{bmatrix} \theta' = J\theta' \tag{5}$$

From this we can extrapolate the change in position caused by a change in joint angles.

$$\Delta S \approx J \Delta \theta \tag{6}$$

The Jacobian matrix has two components, one for the linear velocity J_v and one for the angular velocity J_ω. From equations 3 and 4 we can factor out θ' to solve for J.

$$J_v = \big[z_i \times (u - p_i)\big]_{3 \times n} \tag{7}$$

$$J_\omega = \big[z_i\big]_{3 \times n} \tag{8}$$

The linear component of the Jacobian for each joint is the cross product of the z-axis of that joint z_i and the vector from the joint position to the end effector position $(u - p_i)$. The angular component of the Jacobian for each joint is just the z-axis of that joint z_i. Note that the Jacobian depends on the current configuration of the manipulator and needs to be recalculated each time the joint angles change. For a more thorough explanation of the Jacobian and its derivation for manipulators that do not follow the Denavit-Hartenberg convention, see [4].

D. Singular Value Decomposition

The Singular Value Decomposition (SVD) [5] decomposes any matrix into three operations. A rotation, scale, and another rotation. This decomposition exists for all matrices.

$$A = U \Sigma V^T \tag{9}$$

Where U is an orthonormal matrix, Σ is a diagonal matrix where each element on the diagonal $\sigma_i \geq 0$ is a **singular value** of A, and V^T is another orthonormal matrix. The columns of U are the left singular vectors, and the rows of V^T are the right singular vectors. The ratio of the largest σ_i over the smallest σ_i is called the condition number of A.

$$cond(A) = \frac{\max(\sigma)}{\min(\sigma)}$$

The rank of A is equal to the number of nonzero σ_i. A matrix is singular iff some $\sigma_i = 0$, and called near-singular, poorly-conditioned, or ill-conditioned if the condition number is large. The condition number gives a sense of the numerical error when multiplying by A. A higher condition number implies more numerical error.

The SVD of the Jacobian is interesting for two reasons. First, it can be used as a numerically stable method to compute the pseudoinverse.

$$J = U \Sigma V^T$$

$$J^+ = V \Sigma^+ U^T \tag{10}$$

Where $\Sigma^+ = \mathrm{diag}(\frac{1}{\sigma_i})$ for nonzero σ_i and 0 otherwise. Second, the condition number gives a sense of how easy it is for the manipulator to move the end-effector in any direction. The left singular vectors which U contains is a mapping of components of the velocity onto primary orthogonal abstract concepts in the space of linear and angular velocity. The concepts with higher singular values are easier to achieve. Moving in the direction of concepts with very small singular values requires large joint angle changes for a small change in position. We can use these concepts to inform which directions we should move in when near singularities to reduce the necesary change in joint angles. As the condition number approaches infinity, J loses rank and it becomes impossible for the manipulator to be moved in some direction. The left singular vectors, sorted in descending order by their associated singular values, form a basis where each consecutive direction is harder to move in.

E. Setting a Desired Transform: Lie theory

Given homogeneous transformation matrices for the current pose T_c and desired pose T_d, we must solve for $s = \begin{bmatrix} v & \omega \end{bmatrix}^T$ which will transform our current pose into the desired pose. s must be a velocity instead of a rotation matrix as we can interpolate a velocity to achieve some partial transformation. Lie theory provides exactly that [6]. Lie theory relates the manifold of homogeneous transformation matrices with vectors in $\mathbb{R}^6$. This paper seeks to cover the minimal amount of Lie theory to solve for s. For a more through treatment, derivation of the equations below, and intuition behind the abstract theory, see [6] or [7].

The **hat operator** $(*)^\wedge$ maps a 3×1 vector to a skew-symmetric matrix:

$$\omega = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \quad \omega^\wedge = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{11}$$

$\omega^\wedge$ is skew-symmetric as $-\omega^\wedge = (\omega^\wedge)^T$. The **vee operator** $(*)^\vee$ performs the inverse.

$$\begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}^\vee = \omega \tag{12}$$

The $\log$ map defined as the matrix logarithm takes a rotation matrix R to a skew-symmetric matrix $\omega^\wedge$.

$$\log(R) = \omega^\wedge \triangleq \frac{\theta(R - R^T)}{2\sin\theta} \tag{13}$$

Where $\theta = \cos^{-1}(\frac{\mathrm{Tr}(R)-1}{2})$. θ is the rotation angle about the euler-axis of the rotation matrix. Note that the equation 13 is indeterminate when $\mathrm{Tr}(R) = 3$, i.e. when R is the identity matrix. $\omega = [0]_{3\times1}$ in this case.

$$\omega = \log(R)^\vee = \frac{\theta(R - R^T)^\vee}{2\sin\theta} \tag{14}$$

ω here can be interpreted as angular velocity. It defines an axis $n = \frac{\omega}{|\omega|}$ and an angle of rotation $\theta = |\omega|$. We must define a separate addition operator which induces the rotation defined by ω to our rotation matrix.

$$R_d = RR_c = \omega \oplus R_c \triangleq \exp(\omega^\wedge)R_c \tag{15}$$

$$\exp(\omega^\wedge) = I + \frac{\sin\theta}{\theta}\omega^\wedge + \frac{1-\cos\theta}{\theta^2}(\omega^\wedge)^2 \tag{16}$$

From the above equations, we have been able to define an angular velocity ω and a way to apply that angular velocity to achieve a desired rotation. We can define a similar subtraction operator to solve for the angular velocity ω in the base frame given our current and desired rotations.

$$\omega = R_d \ominus R_c \triangleq \log(R_d R_c^{-1})^\vee \tag{17}$$

Note $R_c^{-1} = R_c^T$. One approach to set $s = \begin{bmatrix} v & \omega \end{bmatrix}^T$ is to solve for ω from the rotation matrices extracted from the homogeneous transforms with equation 17 and to use the difference in position as v. This approach is acceptable and equivalent to the approach below, but we can lean further on Lie theory to solve for both in one equation. To do so, we expand the definition of the $\log$ map to map between homogeneous transforms and 6-vectors containing linear and angular velocity.

To review the notation from earlier:

$$T = \begin{bmatrix} R & u \\ 0 & 1 \end{bmatrix}$$

$$s = \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ x' \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

Now we can define the $\log$ map:

$$\log(T) = \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{18}$$

$$\omega = \log(R)^\vee \tag{19}$$

$$v = V^{-1}u \tag{20}$$

Where V^{-1} has the closed form expression:

$$V^{-1} = I - \frac{1}{2}\omega^\wedge + \frac{(1 - \frac{\theta\cos(\theta/2)}{2\sin(\theta/2)})}{\theta^2}(\omega^\wedge)^2 \tag{21}$$

We can now define a similar subtraction operator to solve for s given desired T_d and current T_c.

$$s = T_d \ominus T_c \triangleq \log(T_d T_c^{-1}) \tag{22}$$

III. Methods for Differential Inverse Kinematics

Now that we can calculate the Jacobian matrix and the desired linear and angular velocities, we need to solve the IK problem for the change in joint angles which achieves that velocity. This section covers three methods, Jacobian transpose, pseudoinverse, and Damped Least Squares (DLS). Jacobian transpose is quick to compute yet leads to high error in the resulting trajectory. Pseudoinverse reduces error by computing the least squares solution but is slower to compute. DLS improves on the pseudoinverse by damping solutions for better behavior near singularities but requires tuning a damping parameter.

Each of these methods computes the Jacobian for the current configuration and uses it to inform the change in joint angles required to stay on some desired velocity. As each of these methods relies on the Jacobian they all provide approximate solutions. A discussion on how to scale the step size to not introduce too much error follows the presented methods. Additionally, we need to close the loop by evaluating the current position through our forward kinematics function instead of assuming the change in position was equal to the scaled step. We end with a discussion on how to design an algorithm which uses any of these methods to control the manipulator and check for convergence at the desired position.

A. Jacobian Transpose

The Jacobian Transpose method is a simple method which aims to solve differential IK in a computationally efficient manner. The base idea is to use the transpose of J instead of the inverse.

As $J\theta' = s$ (5), we get an approximate solution by setting $\theta' = \alpha J^T s$. This creates the following approximation:

$$\alpha J J^T s \approx s \tag{23}$$

This introduces error as J^T is obviously not J^{-1}. The justification to approximate θ' by $\alpha J^T s$ is from the following theorem [2].

Theorem 1. $JJ^T s \cdot s \geq 0$

Proof. $JJ^T s \cdot s = J^T s \cdot J^T s = ||J^T s||^2 \geq 0$ □

This implies that s and $JJ^T s$ point in the same general direction. This method takes many small steps in the correct general direction to converge on the target pose. We must therefore scale each step by an appropriately small $\alpha > 0$ to guarantee convergence. An appropriate way to pick α is to minimize the least squares error, assuming the end effector moves by exactly $JJ^T s$ which we have established is not quite true.

Theorem 2. $\alpha = \frac{s \cdot JJ^T s}{||JJ^T s||^2}$ *minimizes* $||\alpha JJ^T s - s||^2$

Proof. $||\alpha JJ^T s - s||^2 = (\alpha JJ^T s - s)^T (\alpha JJ^T s - s)$
$= ||\alpha JJ^T s||^2 - 2\alpha (JJ^T s)^T s + ||s||^2$
$\frac{d}{d\alpha}(||\alpha JJ^T s||^2 - 2\alpha (JJ^T s)^T s + ||s||^2) = 0$
$\implies 2\alpha ||JJ^T s||^2 - 2(JJ^T s)^T s = 0$
$\implies \alpha = \frac{(JJ^T s)^T s}{||JJ^T s||^2}$ □

Combining this with (23) yields the full value for θ':

$$\theta' = \frac{s \cdot JJ^T s}{||JJ^T s||^2} J^T s \tag{24}$$

B. Pseudoinverse

The pseudoinverse method simply inverts J to solve for θ'.

$$\theta' = J^+ s \tag{25}$$

J^+ is the well known Moore-Penrose inverse of J. It has the qualities of equaling J^{-1} if J is square and of full rank and being the unique matrix which minimizes the least squares error $||JJ^+ s - s||^2$ otherwise. It is preferred to find the pseudoinverse of J over inverting it directly as the computation of the pseudoinverse using the SVD is numerically stable near singularities and it applies to rectangular or singular J. That said, J very infrequently enters true singularities due to numerical error. The primary drawback of the pseudoinverse method is performance near singularities.

When the end effector is near a singularity, there is some direction provided by the left singular vectors which requires a large change in joint angles to achieve a small change in position. The pseudoinverse method as written is agnostic of these near-singularity problems and does not attempt to avoid them or move in easier directions. Therefore, we will freqently see large changes in joint angles as we follow the naive shortest path to the desired pose. As mentioned before, one sensible way to compute the pseudoinverse is through the SVD [5].

C. Damped Least Squares

Damped Least Squares (DLS) attempts to resolve the poor performance of the pseudoinverse method near singularities by adding a regularization term. The DLS method minimizes:

$$||J\theta' - s||^2 + \lambda^2 ||\theta'||^2 \tag{26}$$

For some damping constant λ. We can rearrange the above equation as follows.

$$= ||J\theta' + s||^2 + ||\lambda I \theta'||^2 = || \begin{bmatrix} J \\ \lambda I \end{bmatrix} \theta' - \begin{bmatrix} s \\ 0 \end{bmatrix} ||^2 \tag{27}$$

This is now in the typical least squares framework, we can write out the normal equations and solve for θ'.

$$\begin{bmatrix} J \\ \lambda I \end{bmatrix}^T \begin{bmatrix} J \\ \lambda I \end{bmatrix} \theta' = \begin{bmatrix} J \\ \lambda I \end{bmatrix}^T \begin{bmatrix} s \\ 0 \end{bmatrix} \tag{28}$$

This reduces to:

$$\begin{gathered} (J^T J + \lambda^2 I)\theta' = J^T s \\ \theta' = (J^T J + \lambda^2 I)^{-1} J^T s \end{gathered} \tag{29}$$

Setting $\lambda = 0$ reduces 29 to the pseudoinverse method. This illustrates the concept that DLS is a direct extension of the pseudoinverse method. It is unclear how to set a good value of $\lambda > 0$. Larger values of λ improve behavior near singularities as the large joint angle changes required to move in near-singular directions are penalized to favor the smaller changes necessary to move in other directions. Larger values of λ also slow convergence on paths with well-conditioned Jacobians. The value of λ should be picked to reflect how often the end effector will be in a near-singularity configuration.

This method has the same advantages as the pseudoinverse method relative to the Jacobian transpose method. DLS will have faster convergence and better behavior near singularities, yet it is slower to compute and requires tuning the λ parameter.

D. Setting an Appropriate Step Size

The Jacobian transpose, pseudoinverse, and DLS all rely on the Jacobian matrix. The Jacobian is a linear approximation of the forward kinematics function at the current configuration. For each of these methods, we solve for some $\Delta\theta$ which will move the end effector along the desired velocity, assuming $J\Delta\theta = s$. As this assumption relies on a linear approximation, it fails for large s. This can result in large error on the resulting pose, and can even cause the end effector to move in the wrong direction. We must scale the step size down to an appropriate level so our assumption approximately holds.

A first approach to scaling the step size is to apply scaling directly to s. In this approach, we normalize s by setting it equal to $\frac{s}{||s||}$, and then pick some scalar α to set the exact step size.

$$s = \alpha \frac{s}{||s||} \tag{30}$$

Conceptually, this approach calculates some nearby point in the direction of s and takes a step there instead of to the final desired pose. We must also be careful to not apply this scaling when the final pose is before the calculated point so we do not overshoot the end effector, causing it to oscillate around the final position instead of converging to it. This approach does not take into account the magnitude of joint angle changes needed to get to that intermediate point, so some nearby points in Cartesian space will be further away in joint space than others. Running a controller using this method requires calculating the amount of time the manipulator needs to achieve each next desired configuration as it will vary from waypoint to waypoint.

A second approach is to directly scale the change in joint angles. We set some $\Delta\theta_{\max}$ and then multiplicatively scale the maximum θ' to that value. We also take care to check if the maxmimum value in θ' is less than $\Delta\theta_{\max}$ to prevent the oscillation problem described above.

$$\theta' = \theta' \frac{\Delta\theta_{\max}}{\max(\theta')} \tag{31}$$

The advantage of this approach is we can set this $\Delta\theta_{\max}$ to some physical constant, such as the maximum speed for the joint changes, and then solve for the time a trajectory will take. This reduces the number of magic constants we need to choose, and we know each waypoint we generate is less than some maximum distance in joint space. This also simplifies the controller as the waypoints are equidistant in joint space. Knowing the speed of the joints is sufficient to know the time the manipulator needs to achieve the next configuration.

This second method of scaling is also more in spirit with differential inverse kinematics. This is equivalent to solving the desired joint velocities, running the manipulator on those velocities for a short while, before calculating the next joint velocities to maintain the desired trajectory. We are directly controlling velocities in this method instead of calculating intermediate waypoints.

E. Controlling the End Effector

As we discussed in the previous section, we scale each step the end-effector takes down to an appropriate level to reduce the error introduced by the linear approximation from the Jacobian. We now know how to use any of our methods to calculate a single step of the joint angles to bring the end effector along some desired velocity. The algorithm to control the end effector to continue along this velocity iterates over calculating the Jacobian, calculating the scaled change in joint angles, and applying that change. We must recalculate the Jacobian with the current joint angles to close the loop on control.

Algorithm 1 Control Using Differential Inverse Kinematics

```
Input: Desired Transform $T_d$, Tolerance $\epsilon$, Max Change $\Delta\theta_{\max}$
$\theta \leftarrow$ currentJointAngles()
$T_c \leftarrow$ transformFromAngles($\theta$)
$\delta \leftarrow ||T_c - T_d||$
while $\delta > \epsilon$ do
    $s \leftarrow T_d \ominus T_c$
    $J \leftarrow$ jacobian($\theta$)
    $\Delta\theta \leftarrow$ differentialIK($s, J$)
    if $\max \Delta\theta > \Delta\theta_{\max}$ then
        $\Delta\theta \leftarrow \Delta\theta(\frac{\Delta\theta_{\max}}{\max \Delta\theta})$
    end if
    $\theta \leftarrow \theta + \Delta\theta$
    setJointAngles($\theta$)
    $\delta \leftarrow ||T_c -$ transformFromAngles($\theta$)$||$
    $T_c \leftarrow$ transformFromAngles($\theta$)
end while
```

The control algorithm must check for convergence. The first idea of a convergence check is to calculate the distance between the current pose and desired pose, stopping if it is below some threshold. This has the flaw that some desired poses may be unreachable. The control algorithm in this case will oscillate around some minimum distance point to the desired pose, never terminating. A more desirable convergence check is to instead compute the difference between a previous and current pose. This solves the oscillation problem and causes the control algorithm to terminate either when the desired pose is reached or when it gets as close as possible to the desired pose.

Algorithm 1 presents a complete algorithm for control which directs the end effector along the shortest Cartesian path to a desired pose, handling all the discussed considerations. A controller which achieves auxiliary goals, such as collision avoidance, would only have to change the way the velocity s is calculated to incorporate these constraints.

IV. Conclusion

Differential inverse kinematics is an incredibly powerful method. It allow us to set desired velocities for the end effector, enabling us to achieve goals such as minimizing time taken to reach a desired pose or avoiding obstacles. Additionally, the Jacobian provides further insight into the numerical error and the ease of motion of the end effector. Differential inverse kinematics requires prerequisite knowledge on kinematics, linear and angular velocities, the Jacobian, the SVD, and some basic Lie theory to solve for the desired velocity. We presented three methods to solve differential inverse kinematics; Jacobian transpose is computationally efficient yet introduces error on the trajectory and has poor convergence. The pseudoinverse method finds the least squares solution to reduce trajectory error and improve convergence time yet takes longer to compute, and the DLS method introduces a damping parameter to improve upon the pseudoinverse method's performance near singularities at the cost of convergence time out of singularities.

There were many practical considerations for using these methods. We needed to take smaller step sizes to reduce the error introduced by the linear approximation by the Jacobian, we needed to incorporate these scaled steps into a controller which iteratively calculated the next step to take, and we needed to define an appropriate convergence check.

Python implementations of each of these methods and the controller in Algorithm 1 is available online at https://github.com/PaperXLV/DifferentialInverseKinematics.

References

[1] S. Chan and P. Lawrence, "General inverse kinematics with the error damped pseudoinverse," in *1988 IEEE International Conference on Robotics and Automation Proceedings*, Apr. 1988, 834–839 vol.2. DOI: 10.1109/ROBOT.1988.12164.

[2] S. R. Buss, "Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods," en.

[3] S. R. Buss and J.-S. Kim, "Selectively Damped Least Squares for Inverse Kinematics," en, *Journal of Graphics Tools*, vol. 10, no. 3, pp. 37–49, Jan. 2005, ISSN: 1086-7651. DOI: 10.1080/2151237X.2005.10129202. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/2151237X.2005.10129202.

[4] D. E. Orin and W. W. Schrader, "Efficient Computation of the Jacobian for Robot Manipulators," en, *The International Journal of Robotics Research*, vol. 3, no. 4, pp. 66–75, Dec. 1984, ISSN: 0278-3649, 1741-3176. DOI: 10.1177/027836498400300404. [Online]. Available: http://journals.sagepub.com/doi/10.1177/027836498400300404.

[5] W. Gander, "The Singular Value Decomposition," en, Dec. 2008.

[6] J. Solà, J. Deray, and D. Atchuthan, "A micro Lie theory for state estimation in robotics," en, *arXiv:1812.01537 [cs]*, Dec. 2021, arXiv: 1812.01537. [Online]. Available: http://arxiv.org/abs/1812.01537.

[7] J. L. B. Claraco, "A tutorial on SE(3) transformation parameterizations and on-manifold optimization," en, *University of Malaga, Tech. Rep*, p. 68, Apr. 2022.